A Modular Kernel Architecture For Embedded Systems

A Thesis Submitted
in Partial Fulfillment of the Requirements
for the Degree of
Master of Technology

by
Kshitiz Krishna

to the
Department of Computer Science & Engineering
Indian Institute of Technology, Kanpur
December, 1997

CSE-1997-M-KRI-MOD

Entered in system
NISHA
6-4-98

Certificate

Certified that the work contained in the thesis entitled "*A Modular Kernel Architecture For Embedded Systems*", by Mr.*Kshitiz Krishna*, has been carried out under my supervision and that this work has not been submitted elsewhere for a degree.

(Dr. Rajat Moona)
Associate Professor,
Computer Science,
IIT Kanpur.

(Dr. Deepak Gupta)
Associate Professor,
Computer Science,
IIT Kanpur.

December, 1997

Abstract

With the growing application of embedded controllers, there is a need for improving software design processes for embedded systems. As new ideas and technology crop up everyday, the time taken to design and market the product becomes a crucial factor for their success. Moreover the reliability of design is another key issue for embedded software. Since change or modification is difficult in the system, once it is installed, it must go on consistently throughout the life of the system, handling by itself any exceptions that might occur during execution. Another major issue with the design of embedded applications is the non-availability of proper platforms to develop these applications, since traditional operating systems are too bulky and inefficient to be used for these applications.

Traditional design of embedded systems is done right from scratch, mostly using assembly language. This not only increases the complexity of design but also makes the system more error prone and hard to debug. In this thesis we propose the design of a kernel architecture which could be used as a platform to develop these applications. It is a highly reconfigurable kernel and can be used as a platform for development of a wide variety of embedded applications.The kernel is modular and its components are pluggable, which facilitates its use in small as well as large embedded systems. The advantages of using this will be the ease in development of the embedded application which in turn would reduce application development time and increase the reliability of the application by reducing error probability.

Acknowledgments

On the outset I want to thank **Dr. Rajat Moona** for his support and guidance throughout my work. Inspite of his various involvements he was always available and patient. I am also greatly indebted to **Dr. Deepak Gupta** for his guidance and support. I wish to thank my family for encouraging me to go for post-graduate studies. I would also like to thank my friends Atul Kumar, Sameer Goel and Sameer Shah who had helped me with lots of ideas during the design phase.

My gratitude goes to all of my batch-mates, who made my stay here, in the lab as well as in IITK, a memorable one. And last, but most certainly not the least, I acknowledge the M.Tech 1997 batch for the support they have lent me during our short meetings.

Contents

List of Figures

Chapter 1

Introduction

With the development of electronic technology, the cost of processors and interfacing devices is rapidly reducing. This has resulted in a trend towards automation in almost every application ranging from elevators, washing machines and cameras to automated manufacturing plants, missiles, airplanes and spaceships. *Computer systems that are part of such larger systems and control those systems are called embedded systems*[Zal93].

Traditionally the software for these embedded systems is written in assembly language and includes low level intricacies like startup, scheduling of jobs to be performed, interrupt management, etc. This makes the design of an embedded system extremely complex, time consuming and error prone. Further a system thus developed can not be ported to other processors easily. This results in an unwillingness to change hardware to remain in tune with current technology. Existing Operating systems do provide most of these functions. However, general purpose operating systems cannot be used here since embedded systems have more stringent requirements such as real-time scheduling and need for small sized ROM-able code which general purpose operating systems do not provide.

In this thesis we design an embedded kernel. The aim is to design a platform over which various embedded applications can be developed. The kernel should be generic enough to accommodate applications of vastly varying complexities and at the same time should not introduce extra overheads for the application.

1.1 Motivation

Traditional design of embedded systems is done right from scratch, mostly using assembly language. This process increases design complexity, makes the system more error prone and hard to debug. Further, embedded systems are increasingly being used in complex systems today. With the increased complexity of the embedded software, it is clearly unthinkable to have a software system designed in such a manner. Many embedded operating systems handle this situation by providing a large re-usable portion of the software, i.e., the operating System, and let the designer concentrate only on the applications. However such operating systems do leave much to be desired. They most often carry unnecessary baggage and most often are usually not suited for all kinds of applications. Our motivation for this work is to come up with a modular design for embedded systems to overcome many deficiencies in the embedded system design process.

A proper operating system reduces the complexity of design and also speeds up the development of the application which is very crucial in todays market. Apart from this the reduced complexity along with the use of well tested modules greatly enhances the reliability of the system. In this work, we therefore develop a modular way of creating the embedded operating system.

1.2 Existing Embedded Operating Systems

There are many different embedded kernels available[ES] commercially as well as non-commercially. Most of them are proprietary in nature. All of these are targeted to applications of a fixed range of complexities. We provide a brief description of some of these operating systems here. Some of these provide complete compatibility with one or more industry standards like micro-ITRON[uIT93], POSIX[POS92] etc.

1.2.1 VxWorks

VxWorks[VxW] is a commercial operating system designed for use in embedded and real-time applications. Three highly integrated components are included with

VxWorks: a high performance scalable real-time operating system which executes on a target processor; a set of powerful cross-development tools which are used on a host development system; and a full range of communications software options such as Ethernet or serial line for the target connection to the host.

VxWorks' high popularity in the industry can be attributed to its supporting a wide range of industry standards including POSIX 1003.1b Real-Time Extensions, ANSI C (including floating point support) and TCP/IP networking.

VxWorks has been used in many successful projects including the Mars Pathfinder. The problem with VxWorks is that it is targeted towards large and complex problems. The VxWorks kernel provides many functionalities such as task management, floating point support, dynamic memory management etc. which are not required, and are simply overhead for simple systems such as lift controllers etc.

1.2.2 RTXC–Real-Time Kernel

RTXC[Kerb] is also a field-proven kernel. Though the developers claim it to be an embedded kernel, it is more like a generic real-time kernel and is not very configurable. Like VxWorks it is also targeted for large and complex systems only.

1.2.3 Harmony

Harmony[Har] is a multitasking, multiprocessing operating system for real-time control, developed at the National Research Council Laboratories, Canada. Harmony is a portable, extensible and configurable system. It is portable, both across different target computers (typically assembled from single-board computers), and across different development hosts. Harmony was developed to serve a need at NRC for a flexible system for real-time control of robotics experiments, for the development of experimental robot controllers and for other applications of embedded systems where predictable temporal performance is a requirement.

Since Harmony was developed with robotic applications as its target, it can cater to a comparatively much broader spectrum of embedded applications. However, Harmony comes with a huge overhead for small applications.

1.2.4 Maruti

Maruti[Pro] system is being developed at Department of Computer Science, University of Maryland. The purpose of the Maruti project is to create an environment for the development and deployment of critical applications with hard real-time constraints in a reactive environment. Such applications must be able to execute on a platform consisting of distributed and heterogeneous resources and operate continuously in the presence of faults. The major design goals of the project include real-time operation, fault tolerance and distributivity.

1.2.5 C Executive Real-Time Kernels

C Executive[Kera] provides for a set of system features like interrupt-driven device drivers, real-time clock support, a fully preemptive prioritized task scheduler, and four methods of inter-task coordination: semaphores, events, data queues, and signals. Only main memory is required for operation, although an optional file-system is available.

Though C Executive has good performance on quite a wide range of applications but its extensibility is limited only to the developers of the system as the kernel does not allow plug-ins and the source is not available. As a result the application programmer might have to take care of some requirements of the application which should have been part of the kernel.

1.2.6 RTX (Real-Time Operating System)

RTX[Pag] is a hard-real-time executive with extremely fast task context switch time. It guarantees task response within a specific period of time. It is suitable for hard real-time applications. The current build is for Intel processors running in real mode. Much of the code is in C and could be ported to any platform. Protected mode Intel and PowerPC releases are planned. RTX supports the following features:

- very fast context switch time, within a guaranteed time;
- independent threads scheduled by the kernel;

- intertask communications using object queued requests or events;
- queued object message ports;
- classical semaphore support;
- tasks may sleep or run periodically.

The problem with this system is that it runs only in the non-protected mode and many advanced features like virtual memory etc., cannot be provided. The application cannot use any sort of protection even if it is an important requirement. Hence it is neither suitable for small systems because it carries extra baggage, nor for large applications since the application design becomes very complex and error-prone in absence of protection.

1.3 Design Issues

Our objective is to build a highly configurable, generic and reliable kernel which can be used for the development of a wide range of embedded applications.

Typical embedded application may range from a very simple applications, hardly requiring any service from the kernel, to very complex ones which require advanced services and abstractions from the kernel. If the kernel is to satisfy the requirements for all these applications then it should provide for all the possible advanced features that the application might require. The problem that arises here is that the simple applications must carry extra code corresponding to the advanced features that they do not utilize.

Ideally we would like that the kernel contains no code and every service is added to the kernel as a pluggable module so that the simplest of the applications do not have to carry any extra code. But the kernel has to carry at least the startup code and the code to interface different module with each other and with the application. The following are the objectives set-up while designing this kernel.

- This kernel must be very small in order to conserve space as the application in most cases has be put in a ROM.

- It should provide basic functionalities, so that the level of abstraction for the person working at the application level is not brought down to the intricacies of the hardware.
- The kernel should be generic and scalable so that it can be used for a wide range of applications of different sizes and complexity.
- It should be highly configurable. Regardless of the application complexity there should be no extra baggage and no unnecessary overheads.
- The major portion of the kernel should be hardware independent to make it easy to port the system to a new hardware.
- The behavior of the kernel should be deterministic. It should not go to unpredictable states under any circumstances.
- It should be reliable and fault-tolerant. Once an embedded application is installed, it has to run for the entire life time of the system, so it is necessary that it should have appropriate mechanisms to handle and recover from errors and exceptions. In case of fatal break-downs the system should be able to shut down gracefully.
- Most of the embedded applications have real-time operational requirements. Such a platform should thus be appropriate for real-time applications.
- The kernel should provide a standard and well defined interface for both application programmers and the system developers so that the system may easily be used and be developed even further.

1.4 Organization of the Report

The rest of this thesis is organized as follows. In Chapter 2, we discuss the basic design of the kernel and the relationship between its various components. In Chapter 3 the system components are discussed in detail. The design and implementation of the interrupt manager and the thread manager modules are given in Chapter 4 and

Chapter 5 respectively. In Chapter 6, we describe the design and implementation of a simple test application. Finally Chapter 7 presents the conclusions and discusses the ongoing work suggesting some future extensions.

Chapter 2

Kernel Architecture

Embedded applications range from small systems such as elevator controller, to large systems such as digital cameras, cellular phones etc. Clearly the kernel requirements for such systems will be vastly different. Thus the kernel requirements may range from very simple ones like single process support, single threaded execution, flat memory addressing, static memory allocation support, to more complex and sophisticated ones such as multi-processing, multi-threading, advanced memory management etc. The design of our kernel should be such that it can cater to both ends of the spectrum.

2.1 The Design

The solution suggested in this thesis is to have a highly modular and reconfigurable platform. The kernel consists of two parts, the nucleus or the compulsory portion and the configurable portion. We call this nucleus, the **nano-kernel**. The nano-kernel has to go with every application while the configurable portion can be tied up to the nano-kernel as per the needs. The configurable portion consists of several functionally independent modules. Each of these modules provide a set of services such as thread management, interrupt management etc. The idea is to be able to plug each of these modules in or out irrespective of the other modules present. This also provides the flexibility to support multiple algorithms for the same service type e.g.,

we can have different modules for scheduling like pre-emptive or non-pre-emptive, FCFS or round robin or priority based or real time etc., and any of these may be plugged in as per the requirements of the application.

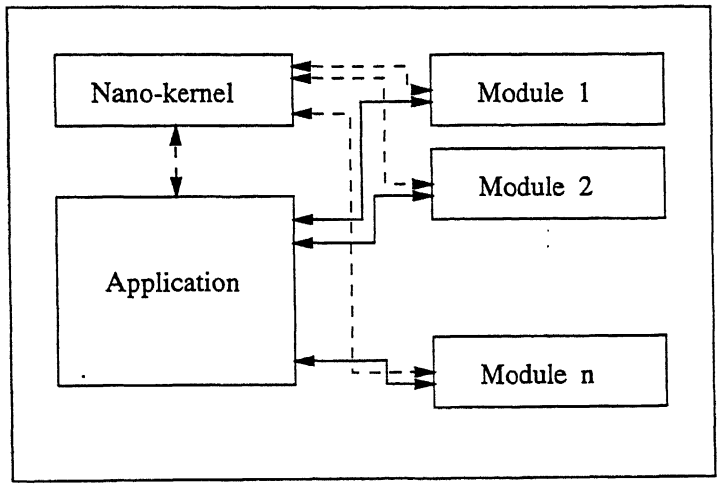

Figure 1: Block Diagram of the System

Figure 1 shows the block diagram of the system. Modules form the configurable portion and may be added or removed from the system as per the requirement at the build time. We will call all such modules as kernel modules.

2.2 The Nano-Kernel

The nano-kernel provides minimal support such as start-up, configuration, attachment and initialization of modules. It is also responsible for providing communication among different kernel modules and between the application and the kernel modules. Whenever a module or application wants a service from another module it can query the nano-kernel about the location of the service-providing routine. If such a module does not exist, nano-kernel provides the default version of the service which is really to do nothing. Otherwise the nano-kernel provides the service entry point of the module. Thus a module can potentially use services from other modules even if it does not exist.

2.3 Kernel Modules

The kernel modules provide various services such as interrupt management, memory management, thread management, process management etc. These modules may be plugged in or out as per the requirements of the application. They might as well use the services of other modules. For this purpose, any module may define its pre-requisites i.e., the modules which should be initialized before initializing these modules. The pre-requisites may be hard or soft. A hard pre-requisite must be present and initialized before this module. Services of such a module cannot be provided by the nano-kernel. In case of soft pre-requisites, the pre-requisite module may or may not be present. If it is not present, nano-kernel provides its services (as default services) but in case the module is present then it must be initialized before this module.

2.4 Inter-module Interface

The kernel provides a standard interface to be used for all service requests between different modules. All inter-module requests are standardized to pass parameters in a parameter block. The results of the services are also returned using the same parameter block. This provides a uniform inter-module interface. In case the kernel supports multi-processing, such an interface is efficient as shared memory pages can be used to prevent the copy of the whole parameter block. This will be applicable even if multiprocessors are used along with shared memory. The parameter block is defined below as a C structure.

```
struct parameter_block{                  // int_2b means integer of 2-bytes
                                         // int_1b means integer of 1-byte
                                         //     or character
        int_2b service_type;             // Service type requested
        int_2b length_to_follow;         // length of data to follow
        int_1b data[];                   // data containing actual
                                         // parameters.
```

```
};
```

2.5 Ports

The communication between modules and between a module and the application is achieved by using an abstraction of ports[1]. Each module is assigned a module id or port statically (we shall use module-id and port interchangeably in this thesis. A module may provide several services each using its own port). The nano-kernel keeps track of the service handler location of a module corresponding to a given port or module id. Whenever a module needs to get a service provided by another module it requests the nano-kernel to give the address of the service providing routine corresponding to the module id or port. The nano-kernel returns the address of the service handler for the requested module. In case the requested module is not present in the given configuration, the nano-kernel returns the address of a default service handler. This handler is incorporated in the nano-kernel to make the different modules independent of the existence of other modules.

2.6 System Configuration

The nano-kernel and all the required kernel modules are put together during configuration which is done by a configuration program statically before building the system. This program takes as input the list of modules to be plugged into the final system and builds up the final system. Here pre-requisites of all the required modules are considered and a order is prepared in which the modules may be initialized such that no module demands a service from a module that is not initialized. This order of initialization is made available to the nano-kernel by the configuration program. The nano-kernel just needs to initialize and link modules in that order, thus bringing up the system.

In this chapter a block design of the kernel has been presented. Here we have

[1]Though these are not exactly ports, the term is being used on the basis of closest similarity.

defined major functionalities of different modules and their relationship with each other. This chapter provides a *bird's eye view* of the design of the kernel. But, the details of the modules, their exact services and their implementation have not been discussed. In the next chapter, we will discuss these details and define the exact interface.

Chapter 3

System Components

As we have seen, components of the system can be categorized as the nano-kernel, the kernel modules, the application and the configuration program. In this chapter, we discuss these components in detail along with their interfaces and services that they need to support.

3.1 Nano-Kernel

The nano-kernel is the component which gets control at system startup. It has two main functions. At startup it initializes all kernel modules that were configured, and later, during the course of execution, it acts as a central exchange where all kernel and application modules contact to query for service handler address of other modules.

3.1.1 Startup Initialization

At startup the nano-kernel initializes all the kernel modules. This involves reading the configuration table and calling each module while passing to it a fixed set of parameters (initialize request parameters). The configuration table is built statically by the configuration program and contains information about the modules present in an appropriate order governed by the pre-requisites of the different modules. Module initialization serves the following purposes.

- The nano-kernel provides its own service handler to a module that it is initializing. This handler is used by the module to get services from the nano-kernel.
- Each module sets-up its startup data structures. For example, the interrupt handler module has to setup the interrupt table.
- Initialization is also used to enquire each module as to which port it will listen to. These port numbers are stored along with the service handler of the concerned module in a port table maintained by the nano-kernel.

3.1.2 Services

The nano-kernel also provides another very important function: that of a central exchange where a module contacts to get the entry point of other modules. For this the nano-kernel provides the following services to other modules.

- **Register** The register function is used to register a given port in the port table and to associate a service handler with that port.
- **Unregister** To remove a port from the port table thus de-activating a kernel module.
- **Query** The query function provides a module, the service handler of another requested module. If the requested module is not present, a default service handler is provided instead.
- **Lookup** This function is used to identify if a given module is present or not in the configuration. Though *Query* function does almost the same job but in-case the module is not present a dummy module address is returned by *Query function*. *Lookup* service is needed by modules which need to know the exact status whether a particular service is present or not and cannot rely on the default services.

Each of these services can be accessed by calling the service handler of the nano-kernel. The address of this routine is provided to each of the modules at the initialization time. The details of the *parameter_block* to be passed to these service

providing routines are given in Appendix B.

One important thing to note here is that the nano-kernel is completely independent of the kernel modules. It knows the service handler for a module associated with a given module id (port) but neither knows nor associates any meaning to the module. For example it might know that handler for a module with module-id 2 is located at physical address 0x4324 but it does not know the functionality associated with module-id 2 or the handler at address 0x4324. More specifically, the nano-kernel has no concept of features like threads, processes, virtual memory etc. It executes in the same manner irrespective of whether these features are present or absent without utilizing, supporting or prohibiting any of these on its own. The only support that it provides is the plug-in concept and expects kernel modules to attach meaning for their services.

3.2 Kernel Modules

Kernel modules provide one or more services to the application or other modules. For example, the memory management module manages (allocates and deallocates) memory for the application and the other modules. A kernel module provides two interfaces described below.

3.2.1 Inter-Module Interaction

Since the set of modules can be very large we cannot provide an exhaustive or even sufficiently large library of these modules. We will only be developing modules which are required by our test applications. The other set of modules whenever required may be developed by the user himself or can be called in from different vendors. This calls for the standardization of the module interface otherwise they cannot be interfaced with the nano-kernel. In our design, modules interact by passing parameter block as given here.

```
struct parameter_block_header{
        int_2b service_type;        //int_2b means a 2-byte integer
```

```
        int_2b length_to_follow;
        };

struct parameter_block_header* module_service_handler
                        (struct parameter_block_header *message);
```

The data in the parameter blocks follows immediately after the parameter block header as shown in Figure 2. Given a parameter block to the service handler, service handler replies are also returned using a similar (sometimes the same) parameter block.

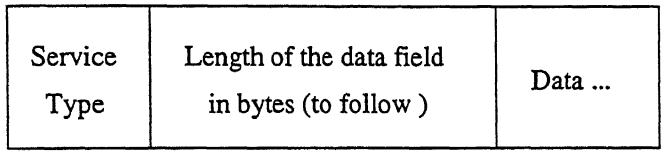

Figure 2: Parameter Block as Data Packet

3.2.2 Initialization Interface

As mentioned earlier, the nano-kernel uses the initialization service of a module at the initialization time. This service type must be supported by every module, irrespective of whether it is needed or not. In case it is not needed by the module it can be a null routine.

Also as a protocol of communication between different modules, it is required by all the modules to keep service type for all the services they provide, an even integer and the reply types as odd integers. The reply type for a successful request completion should be one greater than the service type of the request. Such a protocol is set to make the dummy service handler of the nano-kernel simple. If the nano-kernel gets a request for the service handler's address of a module which is not present then it returns the dummy service handler. The dummy routine increments the service type of any service requesting parameter block that it receives and returns. This gives the requesting module, an illusion of a successful request execution.

Some of the modules that have been designed and implemented are named below. They are discussed in detail later on.

- **The interrupt handler module** which handles and manages all the interrupts and traps. It is discussed in Chapter 4.
- **The thread manager module** which provides support for multiple threads by managing them. It also provides various services related to multiple threads of execution e.g. scheduler, inter-thread synchronization by semaphores signaling between threads etc. It is discussed in Chapter 5.

3.3 Application Module

Application module is the module responsible for carrying out the actual embedded application. The nano-kernel and the kernel modules provide services to the application module. In a general purpose computer there are many and various kinds of applications to be run on the system and hence a way has to be provided to load the applications from disk to the memory by using a loader. In our case a system is designed to carry out only one application which is static and hence the loader is unnecessary. In embedded systems, therefore the application is directly linked to the kernel after the configuration and is executed as part of the kernel itself. Therefore the kernel executes an application by making a call to a subroutine.

But for the kernel to be able to call the application as a subroutine it is necessary to know the location of the application routines. This is done during linking. The nano-kernel expects the application subroutine to have the following prototype.

```
struct parameter_block_header* application_main(
                        struct parameter_block_header* message);
```

This is similar to the `main´ function in C programs.

In case the application needs support of multiple threads and/or processes then the application subroutine that is dispatched by the nano-kernel can be thought of as the first thread and first process. If more threads or processes are required

then the first thread/process of the application must request the kernel through the thread/process module.

3.4 Configuration Program

At startup the nano-kernel needs to know various static parameters determined by a compiler such as which modules are present in the system, in which order should they be initialized, how much space to allocate for the port table etc. In addition to this some modules need to be provided static configuration parameters which are again provided by the configuration program.

The configuration program takes as input a list of modules that are to be linked together. It then configures the individual modules by running their respective configuration programs. Each of these configuration programs generates a C source file containing initialized data variables. These code files are compiled and linked into the final code.

Chapter 4

Interrupt Manager Module

Interrupt management is one of the most important part of any embedded system. One main function of an embedded system is to read the different external inputs to perform certain actions. Most of this input is interrupt driven, i.e., an external device interrupts the processor when it is ready with the data. Since the interrupts can be quite frequent, interrupt handling strategy becomes an important determinant of the system performance.

4.1 The Design

A general approach to interrupt handling is to have a interrupt service routine for each interrupt and have it invoked automatically whenever the interrupt arrives. Conventionally, depending on the interrupt's priority, some or all the interrupts are disabled or queued, when this interrupt service routine is being executed. This approach is inappropriate for cases where a single interrupt might be used for various jobs of different priority. In such cases, if the same interrupt occurs multiple times in succession, fast enough, such that the next interrupt arrives while the previous invocation is being served, then, all subsequent calls are either masked, queued or allowed to be serviced. Each of these cases has its own problems. If the subsequent calls are masked, we lose some input signals which may be urgent. If the calls are queued, the system cannot work since the queue keeps on growing while the system

is not able to serve the requests and if the interrupts are allowed to be serviced, then the stack grows unbounded since the interrupts keep coming but are not able to return.

The approach adapted in the design of this module is to have priorities at the service level rather than at interrupt level. All the interrupts are defined to have equal priority and no interrupt is masked during normal execution. If at any time interrupt masking is required then all interrupts are masked at the same time. The application defines different services to be serviced by a given interrupt along with their priorities. When an interrupt is raised, the default service routine is invoked which executes service routines in order of their priorities. In case another interrupt arises while the previous one is still being served, the interrupt is allowed to be serviced and the previous one is terminated leaving low priority services unserved. Thus this approach favors the execution of high priority services when the interrupt frequency is high and may skip low priority services.

4.2 Pre-requisites for Interrupt Management Module

- **Hard Prerequisites :** None.
- **Soft Prerequisites :** None.

The interrupt manager module has no pre-requisites. The only dependency of this module is a run-time dependency on modules like "virtual memory manager". The interrupt manager at initialization requests the nano-kernel for the *Lookup service*, for the virtual memory manager. If it is present then it additionally sets-up the interrupt tables for the protected mode. Thus the interrupt manager module is generic enough to work under, both, the unprotected environment and protected environment. Since the method of handling the interrupt is dependent on the processor architecture, this module is dependent on the Intel x86 architecture for which it has been implemented.

4.3 Services

The interrupt manager module has to provide certain services in order to let the application and other modules specify their functions to be added or removed from the service routine of a particular interrupt. Apart from the *initialization service* which is mandatory for every module, it provides the following services.

- **Add Handler** - Add a given service subroutine to the list of an interrupt at the given priority.
- **Remove Handler** - Remove a specified service routine or handler from the list of handlers of the specified interrupt.

The details of the *parameter_block* expected by these services are discussed in Appendix B.

4.4 Implementation

The interrupt manager module initializes the interrupt controller and sets up the interrupt table at initialization time. The exact procedure for this is machine dependent. Interrupt table is a table of pointers to the interrupt service routines. The interrupt service routines that are set by the interrupt manager module are small routines which call the *general_interrupt_handler* with an interrupt id *interrupt_number* as parameter. The *general_interrupt_handler* checks if that interrupt is already active or not and if so, returns after setting a *new_interrupt_arrived* flag. If the interrupt was not executing, then the *general_interrupt_handler* invokes various service routines associated with this interrupt in order of their priority. After executing each service routine the *general_interrupt_handler* checks *new_interrupt_arrived* flag and if set, then restarts the execution from the highest priority service routine.

```
general_interrupt_handler(interrupt_number)
1.  Set new_interrupt_arrived(interrupt_number) <- 1.
2.  If interrupt_already_executing(interrupt_number) then return.
3.  Set interrupt_already_executing(interrupt_number) <- 1.
```

```
4.  While new_interrupt_arrived(interrupt_number) do steps 5,6.
5.      Set new_interrupt_arrived(interrupt_number) <- 0.
6.      For all service routines of this interrupt in order of
                        their priority do steps 7,8.
7.              Dispatch routine.
8.              If new_interrupt_arrived(interrupt_number) then
                        go to step 4.
9.  Set interrupt_already_executing(interrupt_number) <- 0.
```

This type of a *general_interrupt_handler* conserves space by preventing the repetition of common code. Also, an important thing to note here is that the *interrupt_manager_module* expects, as configuration parameter, the maximum number of handlers that may be hooked to any one interrupt. If at any time the number of handlers hooked to a particular interrupt exceeds the maximum limit, then the lowest priority handler of the given interrupt is discarded.

Despite our best efforts to keep the code machine independent, some dependencies inherent to processor architecture and the interrupt controller could not be prevented. An effort has been made to reduce the size of the code dependent on the underlying hardware and to increase the portability of the module.

Chapter 5

Thread Manager Module

In our context a thread is an independent flow of execution. Thus while nano-kernel supports a single thread of execution, a thread manager provides multiple threads which may be executing in a system concurrently. This however does not mean that the kernel cannot provide multiple concurrent threads. The important thing is that the threads are transparent to the nano-kernel and other modules. The importance of the thread manager module(TMM) lies at this level of abstraction provided to the application. The TMM provides an application the services related to threads such as thread scheduling, thread creation, thread termination etc. It also provides synchronization services like semaphores and inter thread signaling. Apart from providing these services, the TMM also keeps the existence of multiple threads transparent to the nano-kernel and other modules.

5.1 The Design

To keep the thread manager transparent from nano-kernel and other modules, all the other modules are made thread-unaware. But there is an intimate relationship and to some extent an overlap in the domains of the thread manager and the process manager. Hence, the boundary between the functionalities of the thread manager and the process manager has to be explicitly drawn. Here a thread is the basic execution entity. This is in contrast to the widely used concept of a process being the basic

execution entity. This design was sought since not all applications require process abstractions but the availability of multiple threads provides a better implementation. The thread scheduler is the basic scheduler and the process manager is expected to use the services provided by the thread manager. The process manager is envisaged as a manager of processes which comprise of several threads grouped together and sharing a single address space. Taking such an abstraction for a process is sensible enough since no protection is required in most embedded systems.

The services that the TMM provides can be categorized in three categories.

5.1.1 Basic Thread Management

Basic thread management involves providing the application with the ability to use multiple threads of execution. This includes services to start a new thread, to schedule threads, to terminate a thread and to modify properties of a thread.

There are two common methods to start a new thread.

- The *fork* construct creates a new thread and returns to the statement following the *fork* call in both threads. Here the two thread are considered to have a parent child relationship.
- The *create* construct creates a thread in which control is returned to the specified address. In this case the threads are said to have a peer relationship.

In our design both methods are supported through a single interface. The application program's use is similar to the *create* construct but the thread manager implements it as *fork*. After a fork, the thread manager causes a branch to the specified address if necessary. This simplifies application programming.

Another important component is the scheduling algorithm. In embedded systems different applications require different types of scheduling algorithms. Therefore, TMM design and its data structures make it easy to implement most of the algorithms by just adding a function that determines the next thread to be scheduled. For our application as given in Chapter 6, we have chosen a simple round-robin scheduling algorithm.

A thread is terminated when it exits or is killed by another thread. When a thread terminates the TMM unlocks all the semaphores locked by the terminating thread and clears its thread status structure entry. If its parent thread is still alive then the TMM sends a signal to the parent thread to indicate that a child thread has terminated.

5.1.2 Semaphore Management

TMM also provides a simple semaphore scheme to provide mutual exclusion and synchronize access of shared resources. Although the semaphore service provided by this module is very simple and is similar to binary mutex, it is sufficient for most applications. More sophisticated synchronization services can easily be integrated at the application level itself.

This design is a result of a trade-off between the extra code to be carried along with the TMM and the service provided by the module. Although one of the options could be to provide a rich library of mutual exclusion functions as a separate module but due to its close association and data sharing with thread manager module it was decided to implement it within the TMM to avoid run-time transactions and extra code that may be needed to support functions which may not be used.

5.1.3 Signal Management

Another set of services provided by the TMM include signal management. In embedded systems a very powerful signal management is required since embedded applications have to satisfy real-time constraints. Some signals require a response within a fixed short period. The design here presents a bi-priority signal management scheme. In this scheme, signals can have two level of priorities, the default priority and the high priority. A signal at default priority can be received by the target thread only when it gets scheduled at its turn. In case of high priority signal, the target thread is scheduled out of its turn to let it receive the signal. This scheme guarantees that a high priority signal is received by the target thread as soon as the scheduler gets a chance to schedule it. Even better constraints can be achieved by making the sender

thread voluntarily relinquish control after sending the signal at the application level. This makes the target thread receive the signal almost immediately.

5.2 Prerequisite for TMM

- **Hard Prerequisites :** None.
- **Soft Prerequisites :** Interrupt Manager Module.

The thread manager module needs the interrupt manager module as a soft prerequisite. At initialization the thread manager module requests the nano-kernel to *Lookup* for the interrupt manager module. If the interrupt manager module is present, it attaches the thread scheduler to the 'timer_interrupt' to support pre-emptive scheduling. Otherwise the TMM supports only non-pre-emptive scheduling.

5.3 Services Provided by the TMM

The TMM provides the following services to the application in addition to the *initialization service.* The details of the *parameter_block* expected by these services are discussed in Appendix B.

- **Create Thread** - Set up the stack for the new thread, find an empty entry in the thread array, and make proper entries thereby creating a new thread.
- **Thread Exit** - Mark the current thread status as dead, free all allocated resources and send a signal to parent thread thereby killing the calling thread.
- **Kill Thread** - This service is same as thread exit except that this can kill a specified thread rather than the calling thread.
- **Set Thread Priority** - Set the priority of the specified thread to a specified value.
- **Get Thread Priority** - Returns the priority of the specified thread.

- **Get Thread Id** - Get the thread id of current thread.
- **Suspend Thread** - Suspend the execution of given thread.
- **Resume Thread** - Resume execution of a thread.
- **Thread Switch** - Voluntarily release of control.
- **Thread Wait** - Wait for one or more child threads to die.
- **Thread sleep** - Sleep for a given time.
- **Initialize Semaphore** - Initialize a given semaphore.
- **Set Semaphore** - Locks the given semaphore if not already locked or block if it is already locked.
- **Clear Semaphore** - Unlock the given semaphore and unblock a thread blocked on this semaphore.
- **Close Semaphore** - Close a given semaphore.
- **Set Signal Handler** - Sets a given function as the handler of a given signal in the calling thread.
- **Revoke Signal Handler** - Removes the signal handler which was last set and restore the signal handler to the previous one.
- **Send Signal** - Sends a given signal to a specified thread.

5.4 Implementation

At initialization the TMM sets-up the thread table which involves its initialization and initialization of the scheduler to schedule the first thread. It then requests the nano-kernel to *Lookup* for the interrupt manager. If present then it hooks the *thread_scheduler* to the timer interrupt.

The data structure which stores the thread related variables and is used by this module is given below. Here *int_1b, int_2b and int_4b* represent integers of one, two and four bytes respectively.

```
struct thread_struct {
  int_1b *stack;                /* stack pointer */
  int_2b thread_id;             /* thread id */
  int_2b parent_thread_id;      /* parent thread id */
  enum thread_state status;     /* thread state */
  struct semaphore *pending;    /* semaphore waiting for */
  struct semaphore *allocated;  /* semaphores already locked */
  int_1b priority;              /* thread priority */
  int_1b priority_status;       /* thread priority charge already */
                                /*         paid*/
  struct sig_stat sig;          /* structure giving signal status */
  int_2b sleep;                 /* number of ticks to sleep */
};
```

The variable *allocated* is a pointer to a doubly linked list of *semaphore* structures, which are currently locked by the thread. The *semaphore* structure is given below.

```
struct semaphore{
  struct semaphore *previous;  /* previous in list */
  struct semaphore *next;      /* next in list */
  int_1b status;               /* status of semaphore */
  int_2b thread_no;            /* thread no. which has locked */
                               /* this semaphore */
};
```

The *priority* and the *priority_status* fields of *thread_struct* are not used by the round robin scheduler but can be used by a scheduler that implements priority. The *sig* field of the *thread_struct* is a structure carrying pointers to the signal handlers and there status i.e. the number of times each signal has been received.

The scheduler in the TMM saves CPU registers on the stack and then saves the stack pointer in the thread data structure. It then checks for a thread with pending high priority signal. In case such a thread exists, it is scheduled. This involves restoring the stack pointer, checking for and invoking the concerned signal handlers of this thread and then restoring all the registers from the stack including the instruction pointer. If there exists no thread with pending high priority signal, the scheduling algorithm is invoked which returns the thread to be scheduled next.

Hence we see that the TMM provides a powerful set of services which can even be configured to provide for specific needs. The flexibility it provides makes it usable for the widest range of applications.

Chapter 6

Elevator Controller - An Application Program

The design of the kernel presented in this thesis provides a vastly different method of developing embedded applications which is efficient and reliable. The model kernel that we developed in this thesis along with the interrupt manager and the thread manager module provides a small but sufficient set of services for many applications. In this chapter we discuss the design of a car elevator controller, an application developed using our kernel.

6.1 The Model

Consider the case of a multi-story building built in the heart of the city that has a paucity of parking space. So the parking is decided to be done on terrace or some other *parking floor* necessitating the use of car elevators. Taking such an elevator we put up the following model for our application program.

- The elevator has two gates, one each for entry and exit.
- If a car requires the service of the elevator, a visual sensor at the entry gate informs the elevator controller of the need.

- Under idle condition the elevator cartridge remains at the ground floor with both its doors closed.
- If the elevator is carrying load when some request arrives then it should queue the request.

Further our model elevator has four binary sensors to provide input to the system.

- Sensor at parking floor to check if a car is waiting.
- Sensor at ground floor to check if a car is waiting.
- Sensor to provide the location of the elevator(parking floor or ground floor).
- Sensor to check if a car is in the elevator cartridge or not.

Apart from these sensors there are a few controls that are manipulated by the system depending on the input from the sensors and the current state of the system.

- Open entry gate.
- Close entry gate.
- Open exit gate.
- Close exit gate.
- Elevator control governing the movement of the elevator, viz., move up, move down, stay idle.

6.2 The Design

Applications like an elevator controller require a highly deterministic error handling mechanism. If any error or exception causes the system to go to a non-deterministic or un-handled state, it can be disastrous. Despite the simplicity of the above model it is quite complex to design an application from scratch which can listen and respond to all the requests appropriately and also take care of all the exceptions.

Here we design the application using a much higher level of abstraction. We know that our elevator starts in some unknown state as the cartridge could have had stopped at some undetermined point due to power failure or any other general failure. We first bring the elevator to a known state. This we do by closing all the doors of the elevator and bringing the elevator to the ground floor. We now check if a car is present in the cartridge and if so open the exit door of the cartridge to allow the car to exit. If the car does not move out within some specific time then it is assumed that the car wanted to move to the parking floor and the cartridge is moved to the parking floor. Here also the exit door is opened but the elevator stays till the car moves out. Now we close the exit door and the elevator moves to the ground floor. This brings the elevator to a known state i.e. the cartridge is empty and at ground floor waiting for any requests.

At this point the application is in an initial state and several different threads are created to do various functions of the elevator. In our design we have the following different threads.

- A thread which executes the elevator *finite state machine*(FSM). Depending on the input and the current state of the system, this thread determines the action to be taken, and sends out signals to appropriate threads.

- The thread that controls the main elevator motor. This thread continually tries to move the elevator to its *target location*. *Target location* is a variable modified by different signal handlers of this thread. If the *target location* and the *current location* are same, then this thread keeps the elevator motor idle.

- Thread that polls on all the input sensors. We choose a poll based I/O rather than interrupt driven mainly because the inputs arrive at a very slow rate compared to the processing. This thread informs the FSM thread of any changes that occur in the input sensors.

- A thread to provide timing service. This thread is used to provide time signals to the FSM thread for it to implement delays.

All above mentioned threads work together in coordination to tackle any condition that may arise during the execution of the controller. The design and implementation

of this application took about fifteen man hours. The application is written purely in 'C' language and the size of the source code is about 280 lines.(see Appendix A)

An important point to notice here is that using the kernel we are able to design application at a very high level of abstraction. Thus by using the kernel, design and implementation of the application has become very simple.

Chapter 7

Conclusion

In this thesis we presented a highly modular kernel design for embedded systems. Components of the kernel are pluggable making the kernel highly extensible. The pluggability of different modules allows the small embedded applications to drop out the unnecessary baggage and allows the large ones to add complex services thus making the kernel tailor-able for the widest range of applications. The kernel provides a sufficiently rich set of abstractions eliminating the need for the users to deal with hardware intricacies and details.

The example application of an elevator controller program presented in Chapter 6 and Appendix A clearly confirms the fact that the embedded application development using the embedded kernel presented in this thesis not only reduces the development time but also reduces to a great extent the complexity of design and chances of error.

The kernel design presented here has implemented two modules - the interrupt manager and the thread manager. These two modules are sufficient for small applications. However, for large applications, some more modules such as memory management may be necessary.

7.1 Ongoing Work

The work presented here is a beginning of a large project which targets to improve the design process of software for embedded controllers. Further work is going on to

develop few more modules for the kernel such as physical memory manager module, the virtual memory manager module and the process manager module.

The job of the physical memory manager module is to keep track of the physical memory and to allocate and deallocate memory on request.

The job of the virtual memory manager module is to maintain various virtual memory tables and provide services for allocating, deallocating and changing attributes of the virtual memory segments. It can also provide shared memory segments, protection and handling memory segment faults and exceptions.

The process manager module provides an abstraction of processes each with protected address space. This abstraction is useful, for example when the embedded application consists of different modules coded by different people. In this case we would want processes to remain unaffected by an error in one of the process.

7.2 Future Work

To take the work towards its logical completion the following tasks could be taken up as part of future work.

It is very important to develop more applications using this kernel to verify the correctness and suitability of the kernel. In addition the thread manager module can be modified to provide a powerful inter-thread communication. The kernel should also have a module for secondary storage management. Porting of the kernel and other modules for different hardware architectures and processors is another task that can be taken up. Another important part could be to design a module for demand paging. This will improve the performance of the kernel to a considerable extent. Finally interfacing the kernel with embedded system design tools and other hardware-software codesign tools will greatly ease the rapid development of embedded applications.

Appendix A

Test Application Code

```
/****************************** lift.h ********************************/

struct parameter_block_header* application_main(struct
parameter_block_header* message);
static void open_entry();
static void close_entry();
static void open_exit();
static void close_exit();
static void go_to_ground();
static void go_to_top();
static void motor_control(void);
static void go_up_sig_handler(int_2b count);
static void go_down_sig_handler(int_2b count);
static void go_up();
static void go_down();
static void input_handler(void);
static void init_system(void);
static void cls(void);
static void get_top_car();
static void get_ground_car();
static void timer();
static void set_timer_sig_handler(int_2b count);
```

```
static void reset_timer();

/****************************** lift.c ********************************/

#include<h_lib.h>
#include<libh\libn.h>
#include<libh\libthd.h>
#include"lift.h"

#define CAR_MOVE_TIME 10
#define TOP 9
#define GROUND 0
#define APP_CHILD_THREAD_STACK_SIZE     512

static point_to_func kernel_handler;
static point_to_func thread_manager;

static int_2b motor_thd,timer_thd;
static int_1b motor_target,set_timer;
static int_1b car_inside, at_top, at_ground, moving_up, moving_down;
static int_1b position, entry_open, exit_open;
static int_1b request_from_top;
static int_1b request_from_ground;
static int_1b error;
static int_1b input_handler_stack[APP_CHILD_THREAD_STACK_SIZE];
static int_1b motor_control_stack[APP_CHILD_THREAD_STACK_SIZE];
static int_1b timer_stack[APP_CHILD_THREAD_STACK_SIZE];
static struct semaphore sem;
static int_2b lift_sem;

struct parameter_block_header* application_main(struct
                                    parameter_block_header* message)
{
```

```
kernel_handler=((struct initilized_parameter_block*)
                                    message)->point_to_handler;

/*****************************************************************
  ******* Query the nucleus for the thread handler *******
*****************************************************************/
thread_manager=nucleus_query(kernel_handler,THREAD_H);

/*****************************************************************
 ****************** initilize the system **********************
*****************************************************************/
lift_sem=thd_init_sem(thread_manager,&sem);

init_system();

/*****************************************************************
           Make the timer thread and the input sensor
*****************************************************************/
timer_thd=thd_create(thread_manager,timer_stack,
         APP_CHILD_THREAD_STACK_SIZE,timer,DEFAULT_PRIORITY_THREAD);
thd_create(thread_manager,input_handler_stack,
          APP_CHILD_THREAD_STACK_SIZE,input_handler,
                                        DEFAULT_PRIORITY_THREAD);

/*****************************************************************
 ******* Reset the lift. ie. close all the doors, take  *********
 ******* the cart down, If there is a car in the cart then ******
 ******* open exit door and wait for some seconds. Now close ****
 *******the exit door and if the car is still inside go to*******
 *******top and open exit door. Now start normal loop.  *********
*****************************************************************/
close_exit();
close_entry();
```

```
while(position > GROUND)
{
        thd_sleep(thread_manager,18);
        go_down();
}
if(car_inside)
{
        open_exit();
        thd_sleep(thread_manager,(18*CAR_MOVE_TIME));
        close_exit();
}
if(car_inside)
{
 while(position < TOP)
        {
                thd_sleep(thread_manager,18);
                go_up();
        }
        open_exit();
        while(car_inside);
        close_exit();
}

/*****************************************************************
                    Make the motor thread
*****************************************************************/
motor_target=position;
motor_thd=thd_create(thread_manager,motor_control_stack,
                    APP_CHILD_THREAD_STACK_SIZE, motor_control,
                                        DEFAULT_PRIORITY_THREAD);

/*****************************************************************
call a thread switch so that all the sensor threads get a chance
```

```
**********************************************************************/
thd_switch(thread_manager);

error=0;
while(!error)
{
if((car_inside) && (moving_down || moving_up)) continue;
        if(car_inside)
        {
                open_exit();
                while(car_inside);
                close_exit();
                continue;
        }
        if(at_top && request_from_top)
        {
                get_top_car();
                go_to_ground();
                thd_switch(thread_manager);
                continue;
        }
        if(at_ground && request_from_ground)
        {
                get_ground_car();
                go_to_top();
                thd_switch(thread_manager);
                continue;
        }
        if((at_top || moving_up) && (!request_from_top))
        {
                go_to_ground();
                continue;
        }
```

```
                if(at_ground && request_from_top)
                {
                        go_to_top();
                        continue;
                }
                if(at_ground) continue;
                if(moving_up && request_from_top) continue;
                if(moving_down && request_from_ground) continue;
                if(moving_down && request_from_top)
                {
                        go_to_top();
                        continue;
                }
        }
        return message;
}

static void go_to_ground()
{
       thd_send_signal(thread_manager,3,motor_thd,1);
}

static void go_to_top()
{
       thd_send_signal(thread_manager,2,motor_thd,1);
}

static void motor_control(void)
{
        struct sig_link sig_up,sig_down;

        sig_up.point_to_sig=go_up_sig_handler;
        sig_down.point_to_sig=go_down_sig_handler;
```

```
thd_set_sig_handler(thread_manager,2,&sig_up);
thd_set_sig_handler(thread_manager,3,&sig_down);

while(1)
{
        if(motor_target < position)
        {
                moving_down=1;
                moving_up=0;
                at_top=0;
                at_ground=0;
                go_down();
        }
        if(motor_target > position)
        {
                moving_down=0;
                moving_up=1;
                at_top=0;
                at_ground=0;
                go_up();
        }
        if(motor_target == position)
        {
                moving_up=0;
                moving_down=0;
                if(position == TOP)
                {
                        at_top=1;
                        at_ground=0;
                }
                if(position == GROUND)
                {
                        at_top=0;
```

```
                                at_ground=1;
                        }
                }
                thd_sleep(thread_manager,18);
        }
}

static void go_up_sig_handler(int_2b count)
{
        motor_target=TOP;
}

static void go_down_sig_handler(int_2b count)
{
        motor_target=GROUND;
}

static void get_top_car()
{
        open_entry();
        reset_timer();
        while((!car_inside)&&(set_timer || (!request_from_ground)));
        close_entry();
}

static void get_ground_car()
{
        open_entry();
        reset_timer();
        while((!car_inside)&&(set_timer || (!request_from_top)));
        close_entry();
}
```

```
static void timer()
{
        struct sig_link sig_timer;

        sig_timer.point_to_sig=set_timer_sig_handler;
        thd_set_sig_handler(thread_manager,2,&sig_timer);

        while(1)
        {
                if(set_timer)
                {
                        thd_sleep(thread_manager,(18*CAR_MOVE_TIME));
                        set_timer=0;
                }
                thd_switch(thread_manager);
        }
}

static void set_timer_sig_handler(int_2b count)
{
        set_timer=1;
}

static void reset_timer()
{
        thd_send_signal(thread_manager,2,timer_thd,1);
}
```

Bibliography

[ES] AI Lab Zurich : Links : Embedded and Real-Time Systems. URL. *http://www.ifi.unizh.ch/groups/ailab/links/embedded.html.*

[Har] Harmony. URL. *http://wwwsel.iit.nrc.ca/harmony.html.*

[Kera] C Executive Kernel. URL. *http://www.mcb.net/jmi/cexec.html.*

[Kerb] RTXC-Real-Time Kernel. URL. *http://nps.venture-web.or.jp/Embedded /Rtxc.html.*

[Pag] RTX Page. URL. *http://world.std.com/ mikep/rtx-page.html.*

[POS92] POSIX. System application program interface - amendmment 1: Real time extension. ieee project p1003.4, draft 13. In *Portable Operating System Interface Part 1*, September 1992.

[Pro] The Maruti Project. URL. *http://www.cs.umd.edu/projects/maruti/.*

[uIT93] uITRON. Industrial specifications 3.0 - for micro kernels. In *The Real-time Operating system Nucleus.* http://tron.is.s.u-tokyo.ac.jp/ TRON/ITRON/home-e.html, June 1993.

[VxW] Wind River Systems VxWorks. URL. *http://www.wrs.com/html/vxwks52. html.*

[Zal93] Janusz Zalewski. *Real-time Systems Glossary.* The University of Texas of the Permian Basin, Odessa, TX, December 1993.

CSE - 1997 - M - KRL MCD